What is this “testing” everyone keeps talking about?

Kevin Metcalf (MetcalfKevin@DeAnza.edu)

Slides and sample code at:
deanza.edu/faculty/metcalfkevin/talks.html

What is this “testing” everyone keeps talking about?

Or: “Oh crap, my talk was accepted; I should probably actually learn how to test stuff in Perl and maybe someone else can learn from my mistakes along the way…”

Perl Testing Ecosystem

- Google search for “perl testing” generates 13,100,000 results (in .25 seconds)
- `ok(1, '1 is true');`

re: me and testing

- I’ve been coding in Perl for > 20 years.
- Before 3/20/2015 of this year, I had never written a single test (in any language).
- This talk was accepted on 3/15/2015.
- I will not make any assumptions about your knowledge of testing - including whether it’s useful.

Sample Program

You need to write a program to validate a keycard (or "fob") has access to a specific door.

Program Features

- Program will take two CL args: door num, fob num.
- If not called with exactly two inputs, explain usage.
- If called with a valid door/fob combo, return "Access Allowed".
- If called with invalid door/fob combo, return "Access Denied".
- A "door" will include the building (A..Z), a floor (01..99), and a door number (101..999).
- A "fob" is a 16-digit hex number.

example 001

```perl
#!/usr/bin/perl
use warnings;
use strict;

# UNLESS WE HAVE TWO INPUTS, SHOW DIE WITH USAGE.
if (scalar @ARGV != 2) {
 my $usage =<<"EOT";
Usage: $0 DOORNUM FOBNUM
 DOORNUM is a number of format BF### (BUILDING, FLOOR, NUMBER - e.g. A1101)
 FOBNUM is 16 hex digits.
EOT
 die "\n$usage\n";
}

my $door_number = shift;
my $fob_number  = shift;
print "Validating [$fob_number] has access to [$door_number]...  ";

if (($fob_number eq '0123456789ABCDEF') &&     ($door_number eq 'A1101'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1102'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1103'))
 { print "OK.\n"; }
else { print "ACCESS DENIED.\n"; }
```

example 001

```
[kevin@trggit example001]$ ./fob_access.pl

Usage: ./fob_access.pl DOORNUM FOBNUM
 DOORNUM is a number of format BFFDDD (BUILDING, FLOOR, NUMBER - e.g. A01101)
 FOBNUM is 16 hex digits.

[kevin@trggit example001]$ ./fob_access.pl A01101 0123456789ABCDEF
Validating [0123456789ABCDEF] has access to [A01101]...  OK.

[kevin@trggit example001]$ ./fob_access.pl Q01101 0123456789ABCDEF
Validating [0123456789ABCDEF] has access to [Q01101]...  ACCESS DENIED.

[kevin@trggit example001]$ ./fob_access.pl A01101 0123456789000000
Validating [0123456789000000] has access to [A01101]...  ACCESS DENIED.

[kevin@trggit example001]$
```

A better approach to testing your code…

If only there was a simple way to test our code!

A better approach would…

- Allow us to run all our tests at once.
- Be automated as much as possible.
- Work even if we refactor our code.
- Help ensure new code doesn’t break something that used to work.
- Force us to code in smaller, easier to maintain chunks.
- Etc

TAP

Test Anything Protocol

Sample TAP output…

```
1..2
ok 1 - The variable $a contains the value "4"
ok 2 - $a plus $b = 9
```

example 005

Sample Perl test program

```
#!/usr/bin/perl
use warnings;
use strict;

use Test::More tests => 2;

my $a = 4;
my $b = 5;

is($a, '4', 'The variable $a contains the value "4"');
is($a+$b, 9, '$a plus $b = 9');
```

```
$ perl test_example.t
1..2
ok 1 - The variable $a contains the value "4"
ok 2 - $a plus $b = 9
```

example 006

What happens when a test fails?

```
1    #!/usr/bin/perl
2    use warnings;
3    use strict;
4
5    use Test::More tests => 1;
6
7    my $a = 4;
8    my $b = 99;
9
10   is($a+$b, 9, '$a plus $b = 9');
```

```
$ perl example006/test_example.t
1..1
not ok 1 - $a plus $b = 9
#   Failed test '$a plus $b = 9'
#   at example006/test_example.t line 10.
#          got: '103'
#     expected: '9'
# Looks like you failed 1 test of 1.
```

```
my $a = 4;
my $b = 5;

is($a+$b, 9, '$a plus $b = 9');
```

In module:

```perl
sub add_two {
  my $a = shift;
  my $b = shift;
  return $a+$b;
}
```

In Test Code:

```perl
is(add_two(4,5), 9, 'add_two(4, 5) returned 9');
```

Some Test:More functions:

- `is()`
  ```
  is($a+$b, 9, '$a+$b is 9.');
  ```
- `ok()`
  ```
  ok($a, '$a is true.');
  ```
- `like()`
  ```
  like(mysub($a), qr/right/, 'Got
  expected output from mysub($a)');
  ```

example 001

```perl
#!/usr/bin/perl
use warnings;
use strict;

# UNLESS WE HAVE TWO INPUTS, SHOW DIE WITH USAGE.
if (scalar @ARGV != 2) {
 my $usage =<<"EOT";
Usage: $0 DOORNUM FOBNUM
 DOORNUM is a number of format BF### (BUILDING, FLOOR, NUMBER - e.g. A1101)
 FOBNUM is 16 hex digits.
EOT
 die "\n$usage\n";
}

my $door_number = shift;
my $fob_number  = shift;
print "Validating [$fob_number] has access to [$door_number]...  ";

if (($fob_number eq '0123456789ABCDEF') &&     ($door_number eq 'A1101'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1102'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1103'))
 { print "OK.\n"; }
else { print "ACCESS DENIED.\n"; }
```

Test Driven Development (way oversimplified)

1. Define a feature you want to implement.
2. Define the test cases for the feature.
3. Write just enough code to implement the feature.
4. Re-factor your code if needed.

Program Features

- **Program will take two CL args: door num, fob num.**
- **If not called with exactly two inputs, explain usage.**
- If called with a valid door/fob combo, return "Access Allowed".
- If called with invalid door/fob combo, return "Access Denied".
- A "door" will include the building (A..Z), a floor (01..99), and a door number (101..999).
- A "fob" is a 16-digit hex number.

Where do we start?

1. Create a .pm file to hold your package code:
   e.g., **`Fobaccess.pm`**

2. Create a subroutine for each code section:
   e.g., **`sub validate_data()`**

3. Create a .t file to hold your test code:
   e.g., **`Fobaccess.t`**

4. "Use" your .pm file in your .t file and add your test cases:
   e.g., **`use Fobaccess;`**

example 001

```perl
#!/usr/bin/perl
use warnings;
use strict;

# UNLESS WE HAVE TWO INPUTS, SHOW DIE WITH USAGE.
if (scalar @ARGV != 2) {
 my $usage =<<"EOT";
Usage: $0 DOORNUM FOBNUM
 DOORNUM is a number of format BF### (BUILDING, FLOOR, NUMBER - e.g. A1101)
 FOBNUM is 16 hex digits.
EOT
 die "\n$usage\n";
}

my $door_number = shift;
my $fob_number  = shift;
print "Validating [$fob_number] has access to [$door_number]...  ";

if (($fob_number eq '0123456789ABCDEF') &&     ($door_number eq 'A1101'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1102'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1103'))
 { print "OK.\n"; }
else { print "ACCESS DENIED.\n"; }
```

tests for usage sub

```
sub validate_input()
```

requires: exactly two inputs

- exactly two inputs
- less than two inputs
- more than two inputs

example 008

```
#!/usr/bin/perl
use warnings;
use strict;
use Test::More tests => 1;
use Fobaccess;

my @good_array = ('A01101', '0123456789ABCDEF');
like(Fobaccess::validate_input(@good_array), qr/Correct/,
    'Exactly two inputs for validate_input() as expected.');
```

example 008

```
package Fobaccess;

use warnings;
use strict;

sub validate_input {
if (scalar @_ != 2) {
  my $usage =<<"EOT";
Usage: $0 DOORNUM FOBNUM
 DOORNUM is a number of format BFFDDD (BUILDING, FLOOR, NUMBER - e.g. A01101)
 FOBNUM is 16 hex digits.
EOT
  return $usage;
 }
return 'Correct number of inputs';
}

1;
```

example 008

```
#!/usr/bin/perl
use warnings;
use strict;
use Test::More tests => 1;
use Fobaccess;

my @good_array = ('A01101', '0123456789ABCDEF');
like(Fobaccess::validate_input(@good_array), qr/Correct/,
'Exactly two inputs for validate_input() as expected.');
```

```
$ perl Fobaccess.t
1..1
ok 1 - Exactly two inputs for validate_input() as expected.
```

example 009

```
#!/usr/bin/perl
use warnings;
use strict;
use Test::More tests => 3;
use Fobaccess;

my @good_array = ('A01101', '0123456789ABCDEF');
like(Fobaccess::validate_input(@good_array),
     qr/Correct/, 'Exactly two inputs for validate_input() as expected');

like(Fobaccess::validate_input('only1val'),
     qr/Usage/, 'Less than two inputs fails as expected for validate_input()');

like(Fobaccess::validate_input('3vals', '3vals', '3vals'),
     qr/Usage/, 'More than two inputs fails as expected for validate_input()');
```

```
$ perl Fobaccess.t
1..3
ok 1 - Exactly two inputs for validate_input() as expected
ok 2 - Less than two inputs fails as expected for validate_input()
ok 3 - More than two inputs fails as expected for validate_input()
```

A successful test…

- … Succeeds as expected
- … Fails as expected!

Program Features

- Program will take two CL args: door num, fob num.
- If not called with exactly two inputs, explain usage.
- **If called with a valid door/fob combo, return "Access Allowed".**
- **If called with invalid door/fob combo, return "Access Denied".**
- A "door" will include the building (A..Z), a floor (01..99), and a door number (101..999).
- A "fob" is a 16-digit hex number.

example 001

```perl
#!/usr/bin/perl
use warnings;
use strict;

# UNLESS WE HAVE TWO INPUTS, SHOW DIE WITH USAGE.
if (scalar @ARGV != 2) {
 my $usage =<<"EOT";
Usage: $0 DOORNUM FOBNUM
 DOORNUM is a number of format BF### (BUILDING, FLOOR, NUMBER - e.g. A1101)
 FOBNUM is 16 hex digits.
EOT
 die "\n$usage\n";
}

my $door_number = shift;
my $fob_number  = shift;
print "Validating [$fob_number] has access to [$door_number]...  ";

if (($fob_number eq '0123456789ABCDEF') &&     ($door_number eq 'A1101'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1102'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1103'))
 { print "OK.\n"; }
else { print "ACCESS DENIED.\n"; }
```

tests for access

```
sub test_access()
```

requires: exactly two inputs, a door and a fob

- has less than two inputs
- has more than two inputs
- has two valid inputs - door and fob data
- has two invalid inputs - door and fob data

example 010

```
#!/usr/bin/perl
use warnings;
use strict;
use Test::More tests => 7;
use Fobaccess;

my @good_array = ('A01101', '0123456789ABCDEF');
ok(Fobaccess::validate_input(@good_array),
   'Two inputs expected for validate_input()');

like(Fobaccess::validate_input('only1val'),
   qr/Usage/, 'One input fails as expected for validate_input()');

like(Fobaccess::validate_input('3vals', '3vals', '3vals'),
   qr/Usage/, 'Three inputs fail as expected for validate_input()');

like(Fobaccess::test_access('only1val'),
   qr/Invalid number/, 'One input fails as expected for test_access()');

like(Fobaccess::test_access('3vals', '3vals', '3vals'),
   qr/Invalid number/, 'Three inputs fails as expected for test_access()');

like(Fobaccess::test_access(@good_array),
   qr/Yes/, 'Two valid inputs OK for test_access()');

like(Fobaccess::test_access('not_a_fob', 'not_a_door'),
   qr/No/, 'Two invalid inputs for test_access() fail as expected');
```

example 010

```
sub test_access {
 if (scalar @_ != 2)
  { return "Invalid number of inputs"; }

 my $door_number = shift;
 my $fob_number = shift;

 if (($fob_number eq '0123456789ABCDEF') &&   ($door_number eq 'A01101'))
  { return 'Yes'; }
 elsif (($fob_number eq '0123456789ABCDEF') &&($door_number eq 'A01102'))
  { return 'Yes'; }
 else { return 'No'; }
}
```

```
$ perl Fobaccess.t
1..7
ok 1 - Two inputs expected for validate_input()
ok 2 - Less than two inputs fails as expected for validate_input()
ok 3 - More than two inputs fails as expected for validate_input()
ok 4 - Less than two inputs fails as expected for test_access()
ok 5 - More than two inputs fails as expected for test_access()
ok 6 - Two valid inputs OK for test_access()
ok 7 - Two invalid inputs for test_access() fail as expected
```

example 010

```perl
#!/usr/bin/perl
use warnings;
use strict;
use Test::More tests => 7;
use Fobaccess;

my @good_array = ('A01101', '0123456789ABCDEF');
ok(Fobaccess::validate_input(@good_array),
   'Two inputs expected for validate_input()');

like(Fobaccess::validate_input('only1val'),
   qr/Usage/, 'One input fails as expected for validate_input()');

like(Fobaccess::validate_input('3vals', '3vals', '3vals'),
   qr/Usage/, 'Three inputs fail as expected for validate_input()');

like(Fobaccess::test_access('only1val'),
   qr/Invalid number/, 'One input fails as expected for test_access()');

like(Fobaccess::test_access('3vals', '3vals', '3vals'),
   qr/Invalid number/, 'One input fails as expected for test_access()');

like(Fobaccess::test_access(@good_array),
   qr/Yes/, 'Two valid inputs OK for test_access()');

like(Fobaccess::test_access('not_a_fob', 'not_a_door'),
   qr/No/, 'Two invalid inputs for test_access() fail as expected');
```

bad example 011

```
#!/usr/bin/perl
use warnings;
use strict;
use Test::More tests=>3;
use Fobaccess;

my @good_array = ('A01101', '0123456789ABCDEF');
like(Fobaccess::validate_input(@good_array),
  qr/Correct/, 'Exactly two inputs for validate_input() as expected.');

# SEVERAL MORE TEST CASES HERE! ...

like(Fobaccess::test_access('not_a_fob', 'not_a_door'),
   qr/No/, 'Two invalid inputs for test_access() fail as expected');
```

```
$ perl Fobaccess.t
1..3
ok 1 - Two inputs expected for validate_input()
ok 2 - Less than two inputs fails as expected for validate_input()
ok 3 - More than two inputs fails as expected for validate_input()
ok 4 - Less than two inputs fails as expected for test_access()
ok 5 - More than two inputs fails as expected for test_access()
ok 6 - Two valid inputs OK for test_access()
ok 7 - Two invalid inputs for test_access() fail as expected
# Looks like you planned 3 tests but ran 7.
```

example 011

```
#!/usr/bin/perl
use warnings;
use strict;
use Test::More;
use Fobaccess;

my @good_array = ('A01101', '0123456789ABCDEF');
like(Fobaccess::validate_input(@good_array),
  qr/Correct/, 'Exactly two inputs for validate_input() as expected.');

...

like(Fobaccess::test_access('not_a_fob', 'not_a_door'),
   qr/No/, 'Two invalid inputs for test_access() fail as expected');

done_testing;
```

```
$ perl Fobaccess.t
ok 1 - Two inputs expected for validate_input()
ok 2 - Less than two inputs fails as expected for validate_input()
ok 3 - More than two inputs fails as expected for validate_input()
ok 4 - Less than two inputs fails as expected for test_access()
ok 5 - More than two inputs fails as expected for test_access()
ok 6 - Two valid inputs OK for test_access()
ok 7 - Two invalid inputs for test_access() fail as expected
1..7
```

Program Features

- Program will take two CL args: door num, fob num.
- If not called with exactly two inputs, explain usage.
- If called with a valid door/fob combo, return "Access Allowed".
- If called with invalid door/fob combo, return "Access Denied".
- **A "door" will include the building (A..Z), a floor (01..99), and a door number (101..999).**
- **A "fob" is a 16-digit hex number.**

tests for door validation (format: BFFDDD)

```
sub validate_door_format()
```

requires: exactly one input, the door to check

- Less than one input
- More than one input
- One input with more than 6 chars
- One input with less than 6 chars
- One input with bad (non-numeric) floor data
- One input with bad (non-numeric) door data
- At least one test of: One input with valid data

tests for door validation (format: 16 hex chars)

`sub validate_fob_format()`

requires: exactly one input, the fob to check

- Less than one input
- More than one input
- One input with more than 16 chars
- One input with less than 16 chars
- One input with bad (non-hex) data
- At least one test of: One input with valid data

example 012

```
like(Fobaccess::validate_door_format(), qr/Not enough inputs/,
   'Less than one input fails for validate_door_format() as expected');

like(Fobaccess::validate_door_format('two inputs', 'two inputs'), qr/Extra inputs/,
   'More than one input fails for validate_door_format() as expected');

like(Fobaccess::validate_door_format('A123'), qr/too few/,
   'Too few chars on input fails for validate_door_format() as expected');

like(Fobaccess::validate_door_format('0123456789ABCDEF'), qr/too many/,
   'Too many chars on input fails for validate_door_format() as expected');

like(Fobaccess::validate_door_format('A1234A'), qr/Not a door/,
   'Bad door chars on input fails for validate_door_format() as expected');

like(Fobaccess::validate_door_format('Ab1234'), qr/Not a door/,
   'Bad floor chars on input fails for validate_door_format() as expected');

like(Fobaccess::validate_door_format('A12345'), qr/Valid door/,
   'Good data works for validate_door_format() as expected');

like(Fobaccess::validate_door_format('Z98765'), qr/Valid door/,
   'Good data works for validate_door_format() as expected');
```

example 012

```
like(Fobaccess::validate_fob_format(), qr/Not enough inputs/,
   'Less than one input fails for validate_fob_format() as expected');

like(Fobaccess::validate_fob_format('two inputs', 'two inputs'), qr/Extra inputs/,
   'More than one input fails for validate_fob_format() as expected');

like(Fobaccess::validate_fob_format('0123456789ABCDEF0'), qr/Not a valid fob/,
   'Too many chars on input fails for validate_fob_format() as expected');

like(Fobaccess::validate_fob_format('0123456789ABCDE'), qr/Not a valid fob/,
   'Too few chars on input fails for validate_fob_format() as expected');

like(Fobaccess::validate_fob_format('Z123456789ABCDEF'), qr/non-hex/,
   'Bad (non-hex) data on input fails for validate_fob_format() as expected');

like(Fobaccess::validate_fob_format('0123456789ABCDEF'), qr/Valid fob/,
   'Good data works for validate_fob_format() as expected');

like(Fobaccess::validate_fob_format('ABCDEF0123456789'), qr/Valid fob/,
   'Good data works for validate_door_format() as expected');
```

```
sub validate_door_format {
 if (scalar @_ > 1)    { return "Extra inputs to validate_door_format"; }
 elsif (scalar @_ < 1) { return "Not enough inputs to validate_door_format"; }
 my $input_door = shift;
 if (length $input_door > 6)    { return "Not a valid door; too many chars"; }
 elsif (length $input_door < 6) { return "Not a valid door; too few chars"; }
 unless ($input_door =~ /^[a-z]\d{5}$/i)
  { return "Not a door; does not match BFFDDD"; }
 return "Valid door";
}

sub validate_fob_format {
 if (scalar @_ > 1)    { return "Extra inputs to validate_fob_format"; }
 elsif (scalar @_ < 1) { return "Not enough inputs to validate_fob_format"; }
 my $input_fob = shift;
 if (length $input_fob > 16)    { return "Not a valid fob; too many chars"; }
 elsif (length $input_fob < 16) { return "Not a valid fob; too few chars"; }
 unless ($input_fob =~ /^[\da-f]{16}$/i)
  { return "Not a fob; at least one non-hex char"; }
 return "Valid fob";
}
```

example 012

```
$ perl Fobaccess.t
ok 1 - Two inputs expected for validate_input()
ok 2 - One input fails as expected for validate_input()
ok 3 - Three inputs fail as expected for validate_input()
ok 4 - One input fails as expected for test_access()
ok 5 - One input fails as expected for test_access()
ok 6 - Two valid inputs OK for test_access()
ok 7 - Two invalid inputs for test_access() fail as expected
ok 8 - Less than one input fails for validate_door_format() as expected
ok 9 - More than one input fails for validate_door_format() as expected
ok 10 - Too few chars on input fails for validate_door_format() as expected
ok 11 - Too many chars on input fails for validate_door_format() as expected
ok 12 - Bad door chars on input fails for validate_door_format() as expected
ok 13 - Bad floor chars on input fails for validate_door_format() as expected
ok 14 - Good data works for validate_door_format() as expected
ok 15 - Good data works for validate_door_format() as expected
ok 16 - Less than one input fails for validate_fob_format() as expected
ok 17 - More than one input fails for validate_fob_format() as expected
ok 18 - Too few chars on input fails for validate_fob_format() as expected
ok 19 - Too many chars on input fails for validate_fob_format() as expected
ok 20 - Bad (non-hex) data on input fails for validate_fob_format() as expected
ok 21 - Good data works for validate_fob_format() as expected
ok 22 - Good data works for validate_door_format() as expected
1..22
```

Program Features

- Program will take two CL args: door num, fob num.
- If not called with exactly two inputs, explain usage.
- If called with a valid door/fob combo, return "Access Allowed".
- If called with invalid door/fob combo, return "Access Denied".
- A "door" will include the building (A..Z), a floor (01..99), and a door number (101..999).
- A "fob" is a 16-digit hex number.

example 001

```perl
#!/usr/bin/perl
use warnings;
use strict;

# UNLESS WE HAVE TWO INPUTS, SHOW DIE WITH USAGE.
if (scalar @ARGV != 2) {
 my $usage =<<"EOT";
Usage: $0 DOORNUM FOBNUM
 DOORNUM is a number of format BF### (BUILDING, FLOOR, NUMBER - e.g. A1101)
 FOBNUM is 16 hex digits.
EOT
 die "\n$usage\n";
}

my $door_number = shift;
my $fob_number  = shift;
print "Validating [$fob_number] has access to [$door_number]...  ";

if (($fob_number eq '0123456789ABCDEF') &&     ($door_number eq 'A1101'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1102'))
 { print "OK.\n"; }
elsif (($fob_number eq '0123456789ABCDEF') &&  ($door_number eq 'A1103'))
 { print "OK.\n"; }
else { print "ACCESS DENIED.\n"; }
```

example 012

```perl
#!/usr/bin/perl
use warnings;
use strict;
use Fobaccess;

my $return_value = Fobaccess::validate_input(@ARGV);
if ($return_value ne 'OK')
 { die $return_value; }

if (Fobaccess::test_access(@ARGV) eq 'Yes') {
 print "Access Allowed\n";
}
else {
 die "Access Denied\n";
}
```

```
$ ./fob_access.pl A01101 0123456789ABCDEF
Access Allowed
$ ./fob_access.pl A1 0123456789ABCDEF
Access Denied
```

example 013

```
sub validate_input {
 # UNLESS WE HAVE TWO INPUTS, SHOW DIE WITH USAGE.
if (scalar @_ != 2) {
  my $usage =<<"EOT";
Usage: $0 DOORNUM FOBNUM
 DOORNUM is a number of format BFFDDD (BUILDING, FLOOR, NUMBER - e.g. A01101)
 FOBNUM is 16 hex digits.
EOT
  return $usage;
 }
 my $door_validation_result = validate_door_format($_[0]);
 if ($door_validation_result ne 'Valid door')
  { return $door_validation_result; }
 my $fob_validation_result = validate_fob_format($_[1]);
 if ($fob_validation_result ne 'Valid fob')
  { return $fob_validation_result; }
return 'OK';
}
```

```
$ ./fob_access.pl A01101 0123456789ABCDEF
Access Allowed
$ ./fob_access.pl A1 0123456789ABCDEF
Not a valid door; too few chars at ./fob_access.pl line 8.
```

example 013

```
$ perl Fobaccess.t
ok 1 - Two inputs expected for validate_input()
ok 2 - One input fails as expected for validate_input()
ok 3 - Three inputs fail as expected for validate_input()
ok 4 - One input fails as expected for test_access()
ok 5 - One input fails as expected for test_access()
ok 6 - Two valid inputs OK for test_access()
ok 7 - Two invalid inputs for test_access() fail as expected
ok 8 - Less than one input fails for validate_door_format() as expected
ok 9 - More than one input fails for validate_door_format() as expected
ok 10 - Too few chars on input fails for validate_door_format() as expected
ok 11 - Too many chars on input fails for validate_door_format() as expected
ok 12 - Bad door chars on input fails for validate_door_format() as expected
ok 13 - Bad floor chars on input fails for validate_door_format() as expected
ok 14 - Good data works for validate_door_format() as expected
ok 15 - Good data works for validate_door_format() as expected
ok 16 - Less than one input fails for validate_fob_format() as expected
ok 17 - More than one input fails for validate_fob_format() as expected
ok 18 - Too few chars on input fails for validate_fob_format() as expected
ok 19 - Too many chars on input fails for validate_fob_format() as expected
ok 20 - Bad (non-hex) data on input fails for validate_fob_format() as expected
ok 21 - Good data works for validate_fob_format() as expected
ok 22 - Good data works for validate_door_format() as expected
1..22
```

Program Features

- Program will take two CL args: door num, fob num.
- If not called with exactly two inputs, explain usage.
- If called with a valid door/fob combo, return "Access Allowed".
- If called with invalid door/fob combo, return "Access Denied".
- A "door" will include the building (A..Z), a floor (01..99), and a door number (101..999).
- A "fob" is a 16-digit hex number.

What do I do next?

- Try to modify the code presented today; add tests and write the code for a DB interface instead of **`if/else/elsif`**.
- **`Test::Tutorial`** - Lots of good documentation in there!
- Read up on using the **`prove`** command (and `t/` directories).
- Search YouTube for other YAPC talks on testing.